Memory-Aware Genetic Algorithms for Task Mapping on Hard Real-Time Networks-on-Chip

Lloyd Robert Still and Leandro Soares Indrusiak
Real-Time Systems Group, Department of Computer Science
University of York
York, United Kingdom
Email: lloyd.still@alumni.york.ac.uk, leandro.indrusiak@york.ac.uk

***Abstract*—The problem of mapping hard real-time tasks onto networks-on-chip has previously been successfully addressed by genetic algorithms. However, none of the existing problem formulations consider memory constraints. State-of-the-art genetic mappers are therefore able to find fully-schedulable mappings which are incompatible with the memory limitations of realistic platforms. In this paper, we extend the problem formulation and devise a memory architecture, in the form of private local memories. We then propose three memory models of increasing complexity and realism, and evaluate the impact these additional constraints pose to the genetic search. We conduct extensive experiments using tasks and communications from a realistic benchmark application, and compare the proposed approach against a state-of-the-art baseline mapper.**

***Index Terms*—Real-time systems, Multi-core processing**

I. Introduction

As real-time systems continue to adopt multi-core and many-core processors, end-to-end time-predictability has become a necessity. In hard real-time systems, it is imperative that tasks are guaranteed to meet their deadlines, and this must include both computation over cores as well as communication over on-chip interconnects. This is vital in avoiding the failure of safety-critical features of such systems, which are often found in the automotive, aerospace and health domains.

The *Task Mapping Problem* (TMP) is a crucial stage in the development of any multi- and many-core system. Given a set of n tasks and a set of m cores, a task mapping dictates which tasks are allocated to execute on which processing core out of the m^n different possible mappings [16]. The solution to the TMP greatly influences system performance, determining how inter-core communication is performed. Interconnect architectures such as Networks-on-Chip (NoCs) make the TMP more complex, as system performance can highly depend on communication latency. A poor mapping could result in heavy congestion over the NoC, causing tasks to miss their deadlines, which could manifest as undesired or even unsafe behaviour.

Existing works consider a variety of optimisation metrics when solving the TMP. These include, but are by no means limited to: real-time schedulability, communication latency, energy dissipation, system utilisation, and load balancing [15]. However, these approaches all make critical assumptions regarding the amount of memory available to each local core. Surprisingly, these approaches all rely on the unrealistic and unreasonable assumption that each processing core within a NoC has access to unbounded local memory. The consequences are severe, potentially resulting in the production of mappings that are infeasible – processing cores may require substantially more memory than is physically available.

This paper addresses the application of genetic algorithms (GAs) to evolve task mappings that minimise the overall memory requirement, whilst maintaining the hard real-time schedulability of the system. Previous work has shown that such algorithms are able to evolve fully-schedulable task mappings for hard real-time NoCs, and this paper aims to establish to what extent GAs can cope with an additional dimension over their objective space, namely the amount of memory available to each processing core. To achieve this, we propose suitable extensions to hard real-time analytical models for NoCs, and then integrate them as fitness functions in a GA pipeline. The pipeline will then evolve task mappings in terms of their schedulability and memory requirement.

II. Background

Search algorithms are widely used to statically solve the TMP for NoC platforms. Ascia et al. [2] were among the first to propose a genetic algorithm to minimise energy and average performance of NoC communications. They used system simulation as their fitness function, i.e. as the way to accurately obtain average performance figures for each mapping alternative, and thus guide the genetic search towards optimised results. Their extensive set of experiments provided important insights on the potential of such technique, but also highlighted the heavy cost of performing thousands of simulations, which could take several hours or even days.

Mesidis and Indrusiak [9] focused on genetic algorithms coupled to schedulability models as their fitness function, aiming to find mappings that fulfil hard real-time constraints (i.e. tasks and packets never miss their deadlines). Besides showing the successful application of the technique, they have also shown that schedulability models are more suitable as fitness functions to search heuristics, as they perform orders of magnitude faster than simulation. They can therefore easily be applied to thousands of individuals across hundreds of generations. Following this trend, Ayari et al. proposed the use of custom genetic operators to make better use of schedulability models as search guides in GAs [3], and Nikolic et al. proposed the minimisation of NoC virtual channels by using

schedulability models to guide a simulated annealing heuristic [10]. The possibilities of multi-objective optimisation of search heuristics were investigated in [12], which combines a schedulability model based on [13], and the energy macromodel from [11]. The heuristic was able to evolve task mappings that fulfil hard real-time constraints and simultaneously minimise NoC energy dissipation.

Among the works referenced above, most of them either consider hard real-time constraints over task computation latencies [3] or over NoC communication latencies [9], [10]. Sayuti [12] was the first to consider constraints over end-to-end latencies (i.e. both task computation over cores and packet-based communication over the NoC). None of the approaches consider constraints on the amount of memory available to the cores to store code and data. They assume that local memories are sufficiently large to hold the code of all tasks mapped to a given core, as well as all the data they hold, send and receive.

III. Genetic Task Mapping

Figure 1 depicts the typical genetic algorithm pipeline proposed in [9], which has been extended in several subsequent works. Chromosomes encode task mappings using a list structure, where each gene represents a task, and its value refers to the core that the task maps to. The initial population is randomly generated, and is then diversified by a pipeline of genetic operations (selection, crossover and mutation), producing an offspring population. The offspring population is evaluated according to a fitness function based on a schedulability metric (e.g. the number of fully-schedulable tasks and packet flows [3], [9]). The fittest chromosomes then become a part of (or replace) the parent population for the next iteration, until some termination criteria is met.

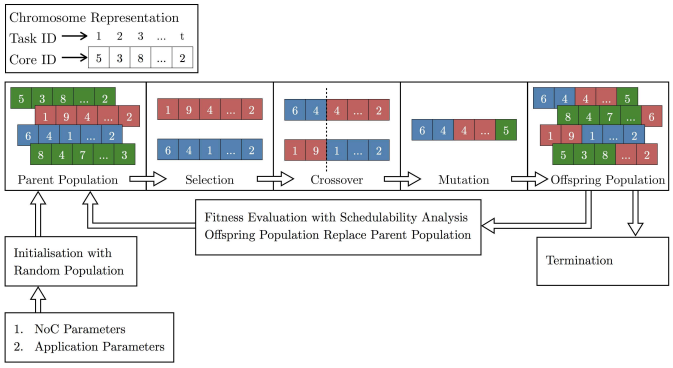

Fig. 1. Chromosome representation and genetic task mapping pipeline.

There are several schedulability analyses for priority-preemptive NoCs [7], [8], [17], predominantly derived from Shi and Burns [13]. These metrics rely on a number of NoC parameters (e.g. router and link latencies, routing protocol, number of virtual channels) and application parameters (e.g. which tasks send and receive packets, time intervals, packet sizes). The analyses each calculate the worst-case end-to-end latency for each packet from its source to its destination, and then compare that value with the respective packet deadline. In a fully-schedulable mapping, all packets are guaranteed to meet their deadlines. Typical fitness functions are based on maximising the percentage of schedulability or minimising the number of unschedulable tasks and packet-flows. Some genetic mappers also consider secondary objectives such as energy minimisation in their fitness functions [12]. The composition of multiple objectives can be achieved through simple weight factors, or the notion of Pareto dominance [5]. In the experiments reported in section V, we adopt the analysis presented in [7], for its convenience and improved tightness. However, we could use any of the previously cited analyses without significantly changing our proposed approach.

IV. Modelling Memory Constraints

This paper extends the state-of-the-art genetic task mappers by making them aware of constraints imposed by the memory subsystem in NoCs. Figure 2 shows a typical 2D-mesh NoC architecture, where the NoC routers P and links Λ interconnect a set of tiles Π, each of them including a processing core and a private local memory. Processing cores use the local memory to store code, data or both. They use the NoC infrastructure to perform block transfers to/from the private local memories of neighbouring cores, or to/from external memories via memory controllers connected to the NoC.

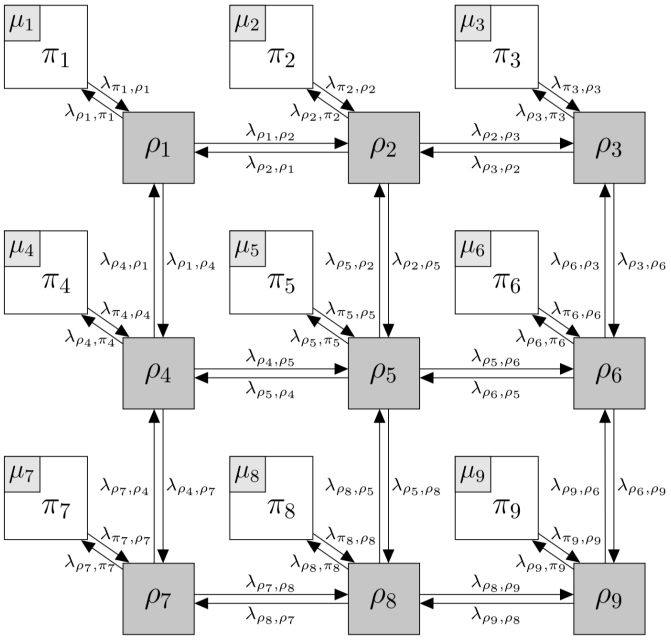

Fig. 2. Typical NoC platform, arranged in a 3×3 2-D regular mesh topology.

As reviewed in Section II, existing genetic task mappers take into account the computational capacity of the core in a given tile when considering mapping tasks onto it. Mapping multiple tasks to the same core will eventually force some tasks to become unschedulable, as they won't have enough time on the core to finish their computations before their deadlines. However, none of the existing mappers takes into account the constraints imposed by the amount of private local memory

available at each tile. By mapping multiple tasks to the same tile, it is possible that there will be insufficient memory to hold the code, data and communications required by the tasks during execution. In order to investigate the impact of such constraints, we propose a number of memory models that can be used as secondary objectives in genetic mapping pipelines. We therefore aim to minimise the amount of memory required by each NoC tile, whilst ensuring the system remains fully-schedulable.

We define a NoC platform Ψ as a set of s homogeneous processing cores $\Pi = \{\pi_1, \pi_2, ..., \pi_s\}$, a set of r routers $\mathrm{P} = \{\rho_1, \rho_2, ..., \rho_r\}$, and a set of unidirectional links $\Lambda = \{\lambda_{\pi_1,\rho_1}, \lambda_{\rho_1,\pi_1}, \lambda_{\rho_1,\rho_2}, \lambda_{\rho_2,\rho_1}, ..., \lambda_{\pi_s,\rho_r}, \lambda_{\rho_r,\pi_s}\}$. An application consists of a set of t tasks $\mathrm{T} = \{\tau_1, \tau_2, ..., \tau_t\}$, which we assume to be either periodic or sporadic. Tasks communicate over the NoC by sending communication messages $\Phi = \{\varphi_1, \varphi_2, ..., \varphi_t\}$, each of them concisely defined by a 3-tuple $\varphi_i = \langle \tau_i^s, \tau_i^d, z_i \rangle$, where τ_i^s and τ_i^d are respectively the source and destination tasks of the communication, and z_i is the size of the communication message, measured in bytes. The function $map(\tau_i) = \pi_\alpha$ is defined as the processing core π_α on which task τ_i is executing. The set of tasks mapped to a specific processing core can be obtained using the inverse function: $map^{-1} : \Pi \rightarrow \mathrm{T}$.

The described architecture and application models are adopted by most of the works reviewed in Section II. In this paper, we require further definitions relating to memory constraints. We define μ_α to denote a block of *private local memory* associated with each processing core $\pi_\alpha \in \Pi$. The private local memories within the platform form a set: $\mathrm{M} = \{\mu_1, \mu_2, ..., \mu_s\}$. The function $cap(\mu_\alpha)$ is used to represent the size (in bytes) of the private local memory available to a processing core π_α. We must distinguish between the amount of private local memory *available to*, and the amount *required by*, a specific core. We therefore also define $req(\mu_\alpha)$ to represent the memory required by a processing core π_α. Since our platform has homogeneous processing cores, the amount of memory available to each core is identical: $cap(\mu_1) = cap(\mu_2) = ... = cap(\mu_s)$.

The memory requirement $req(\mu_\alpha)$ for each processing core is derived using a *memory model* associated with the platform. This model defines what system and application properties contribute to the memory requirement. The properties that will influence the memory requirements are the tasks executing on each core, and the communications they send and receive.

In order for the memory system of a NoC to be *feasible*, the amount of memory required by each core must not exceed the amount of memory available to it. The notion of feasibility allows us to ensure that the task mappings produced have realistic memory requirements. Infeasible mappings may have unrealistic memory requirements when compared with the memory capacities offered by commercially available NoCs [4], [1]. The task mapping search space can be pruned by removing any mappings that are clearly infeasible. In a homogeneous platform, the memory system is feasible if and only if the memory capacity $cap(\mu_\alpha)$ of any one of the processing cores π_α is greater than the maximum memory requirement of all processing cores, as defined below, in Equation 1.

$$feas(\Psi) = cap(\mu_\alpha) \geq \max_{x \in \mathrm{M}} req(x) \tag{1}$$

We also define a utilisation performance metric which, for a given $\mu_\alpha \in \mathrm{M}$, calculates the fraction of memory allocated to the processing core that is actually required for execution:

$$U(\mu_\alpha) = \frac{req(\mu_\alpha)}{cap(\mu_\alpha)} \tag{2}$$

We next propose three memory models which are able to determine the minimum amount of private local memory $req(\mu_\alpha)$ required by each core π_α. Each one relaxes restrictions placed on previous model(s) and therefore the final, most sophisticated model aims to provide the most realistic treatment of memory within our assumed NoC platform.

A. Memory Model A: Receiving

Memory Model A is the simplest memory model of the three we consider, and is founded in the assumption that each task is able to begin execution immediately upon its release. In order for this to be possible, it is necessary that the task has received all data from communications destined for it prior to its release. This implies that each communication it receives must be stored in the private local memory until the task is released, justifying the need for it to be considered in our memory model.

We therefore design Memory Model A to account for the memory required to store incoming communications. We define $recv : \Pi \rightarrow \Phi$ to be the set of communications whose destination tasks are mapped to core π_α, more formally: $recv(\pi_\alpha) = \{\varphi_i \in \Phi \mid map(\tau_i^d) = \pi_\alpha\}$. For a given core π_α, we can calculate the amount of required private local memory $req(\mu_\alpha)$ using Memory Model A as follows:

$$req(\mu_\alpha) = \sum_{\varphi_i \in\, recv(\pi_\alpha)} z_i \tag{3}$$

This model does not consider the memory m_i required by each task τ_i that is mapped to the core. It is assumed both the memory required to store the code executed by the task and any dynamic memory allocation required during execution are negligible in size, and therefore small enough to fit in currently unallocated memory. Since this approach is widely used by the studies reviewed in II, we argue that the omission of m_i is reasonable.

Furthermore, the task model assumes that the communication of a message occurs after execution of the task has finished. We might therefore assume that the communication has a storage requirement of z_i – the size of the outgoing communication. However, without loss of generality, we propose that communications are not generated during execution of the task, but instead are generated on a flit-by-flit basis after execution, and immediately transmitted. The result is that it is not necessary to store the communication prior to sending, alleviating the need to consider the size z_i of outgoing communications in our memory model.

B. Memory Model B: Receiving and Sending

The memory requirements as derived under Memory Model A provide a sound starting point, but are largely unrealistic because of the assumptions made regarding the treatment of outgoing communications. While some tasks may generate their communications using the assumed flit-by-flit method, it is unrealistic to assume all tasks will behave this way. It is likely that most tasks will need to store and process their communication before it is sent.

Memory Model B is an extension of Memory Model A, and adopts a more sophisticated approach to handling communications. Specifically, Memory Model B accounts for the memory required to store both incoming and outgoing communications. We define $send : \Pi \rightarrow \Phi$ to be the set of communications whose source tasks are mapped to core π_α, more formally: $send(\pi_\alpha) = \{\varphi_i \in \Phi \mid map(\tau_i^s) = \pi_\alpha\}$. For a specific core π_α, we can calculate the amount of required private local memory $req(\mu_\alpha)$ using Memory Model B as follows:

$$req(\mu_\alpha) = \sum_{\varphi_i \in\, recv(\pi_\alpha)} z_i \quad + \sum_{\varphi_j \in\, send(\pi_\alpha)} z_j \tag{4}$$

As before, this model does not consider the memory m_i required by each task τ_i that is mapped to the core, and the size of the code and any dynamic memory allocation required for tasks to execute is assumed to be negligible.

C. Memory Model C: Receiving, Sending and Code

The final model, Memory Model C, is the most sophisticated of the three models, and therefore provides the most realistic representation of memory. This model relaxes the assumption made in both of the previously defined models that the memory requirement m_i of each task is negligible. This is an obvious extension – it is reasonable to assume that tasks require both memory to store their code and access to a heap/stack during execution. While some simple tasks such as comparison operators will require very little code and dynamic memory to execute, more complex real-time tasks may have memory requirements in the order of kilobytes, or even megabytes [14].

Memory Model C accounts for incoming and outgoing communications, in addition to the memory requirement of each task. For a specific core π_α, we can calculate the amount of required private local memory $req(\mu_\alpha)$ using Memory Model C as follows:

$$req(\mu_\alpha) = \sum_{\varphi_i \in\, recv(\pi_\alpha)} z_i \quad + \sum_{\varphi_j \in\, send(\pi_\alpha)} z_j \quad + \sum_{\tau_k \in\, map^{-1}(\pi_\alpha)} m_k \tag{5}$$

V. Experimental Work

A. Experiment Setup

We conduct an in-depth scenario-based experiment using the *Autonomous Vehicle Application* (AVA) benchmark, proposed in [14]. The AVA benchmark models several subsystems of an autonomous vehicle, including 39 communicating tasks performing functionality such as navigation control, vibration control and obstacle detection through stereo photogrammetry. Task periods vary between 0.04 and 1 second, and communication volumes vary between 1 and 76 kilobytes.

We use the AVA benchmark as a baseline genetic mapper similar to those reviewed in Section II (referred as SOGA, for Single-Objective Genetic Algorithm), aiming to primarily minimise unschedulable tasks and packet flows. We compare this against the proposed genetic task mapper, which also includes a secondary objective that aims to minimise the amount of local memory required by each NoC tile (referred as MOGA, for Multi-Objective Genetic Algorithm). We assume a homogeneous NoC architecture, so the processing cores and private local memories of all tiles must have identical properties (e.g. the amount of available memory). We use Equation 6 as our secondary objective, which minimises the maximum memory requirement of a given task mapping.

$$O_2 = \max_{x \in \mathrm{M}} req(x) \tag{6}$$

For each of the SOGA and MOGA, we perform an experiment using Memory Model A, Memory Model B and Memory Model C. These combinations form our six experimental runs, which we refer to as: SOGA-A, SOGA-B, SOGA-C, MOGA-A, MOGA-B, and MOGA-C.

The parameters for both the SOGA and MOGA remain identical for the experiments with each of the different memory models, and are shown in Tables I and II, respectively. As discussed, we aim to keep the differences between the SOGA and MOGA minimal, with the only changes being the fitness functions and selection operators that they implement.

TABLE I
EXPERIMENTAL PARAMETERS FOR THE SOGA

Parameter	Value
Population Size	100
Number of Generations	100
Fitness Function	(1) # Unschedulable Tasks and Flows
Selection Operator	Binary Tournament Selection
Crossover Operator	One-Point Crossover
Crossover Probability	0.8 (per individual)
Mutation Operator	Uniform Integer Mutation
Mutation Probability	0.01 (per gene)
Elitism	1 Individual
Benchmark	AVA

TABLE II
EXPERIMENTAL PARAMETERS FOR THE MOGA

Parameter	Value
Population Size	100
Number of Generations	100
Fitness Function	(1) # Unschedulable Tasks and Flows (2) Memory Requirement of the Platform
Selection Operator	Binary Tournament Selection (Dominance) NSGA-II
Crossover Operator	One-Point Crossover
Crossover Probability	0.8 (per individual)
Mutation Operator	Uniform Integer Mutation
Mutation Probability	0.01 (per gene)
Elitism	1 Individual
Benchmark	AVA

We select a population size of 100, large enough to promote a diverse population. The initial population is generated randomly, with each gene selected randomly and independently from a uniform distribution. We allow the algorithm to evolve for 100 generations, with no early stopping criteria. This should provide enough generations for the genetic operators to effectively search the solution space. Previous studies have shown that most GAs are able to find mappings of the AVA benchmark in under 50 generations [6]. However, we argue that with additional constraints, 50 generations may be too optimistic to obtain a schedulable mapping. This is supported by [12], where by considering schedulability and energy dissipation, more generations are required to obtain a schedulable mapping.

The evolutionary process is inherently stochastic – the initial population is randomly generated, the selection operator randomly selects tournament participants, and the crossover and mutation operators are applied probabilistically. Each experiment that implements Memory Model C has an additional random element. Namely, before each evolution, each task is assigned a random memory requirement, chosen uniformly between 2048 and 16384 bytes. To account for this stochasticity, we conduct 100 repetitions of each of the six experimental configurations, over which we average our results.

The parameters of the NoC platform that we use for each experiment are summarised in Table III. All NoC parameters are identical in each experiment with the exception of the memory model, which we set as Memory Model A, Memory Model B, or Memory Model C, depending on the experiment.

TABLE III
EXPERIMENTAL PARAMETERS FOR THE NOC

Parameter	Value
Topology	4×4 2D Regular Mesh
Homogeneity	Homogeneous Cores, Routers and Links
Core Clock Speed	100MHz
Routing Algorithm	XY Static Routing
Switching Mechanism	Wormhole Switching
Arbitration Policy	Priority-Preemptive Arbitration with Virtual Channels and Credit-Based Flow Control
Router Latency	1 cycle
Buffer Size	2 flits
Link Width	1 byte (8 bits)
Link Latency	1 cycle
Network Clock Speed	100MHz
Memory Model	A, B or C

For each of the 100 evolutions, we log certain performance metrics of the population at each generation. To achieve this, for each generation we first calculate the value of each metric on a per-individual basis, before averaging over the population. This process yields a set of 100 independent results that will form the basis of our analysis: providing us with sufficient data to compare the performance of the two genetic algorithms, and examine the effect of our three proposed memory models. We collect the following performance metrics:

- *Mean Memory Requirement*: We keep a record of the mean memory requirement per core according to the respective memory model used in the experiment.
- *Mean Schedulability*: We log the number of unschedulable tasks and flows, that may miss their deadlines.
- *Mean Memory Utilisation*: We log the memory utilisation of each processing core of the NoC using Equation 2.

B. Results

This section presents the experimental results for our six different algorithm configurations. Specifically, we consider the memory requirement, schedulability and memory utilisation. For each of these performance metrics, we analyse the results and suggest potential causes of our findings. The genetic mappers were allowed to run for 100 generations, as we employed no early stopping criteria. The values were obtained by taking the average across 100 repetitions.

1) Memory Requirement: The first aspect we considered was the mean memory requirement of the system. Figure 3 shows how the mean memory requirement of all individuals in the population varied over the course of the evolutionary process. Figure 4 shows the distribution over 100 runs of mean memory requirements for the final population.

We next considered the best individual at each generation, rather than taking the mean value of the entire population. The best individual is defined as the one having the lowest memory requirement in the population. Figures 5 and 6 show the evolution of the best individual in the population, and the distribution of best individuals in the final population over the 100 repetitions, respectively.

In all experiment configurations, the mean and minimum memory requirement of individuals in the population decreased significantly as evolution progressed. By the final generation, the average mean memory requirement for the SOGA-A, SOGA-B, and SOGA-C configurations had decreased by approximately 50kB, 90kB and 100kB respectively – despite the fact that the fitness function did not consider memory when selecting individuals. As expected, the MOGA configurations resulted in substantially lower mean and minimum fitness values by the final generation. In fact, the distributions presented in Figure 4 were so distinct that our attempts to apply Kruskal-Wallis H tests resulted in extremely small p-values.

There were evident differences between the MOGA-A, MOGA-B and MOGA-C configurations in the number of generations it took for the evolution to plateau. It appeared that increasing the complexity of the memory models also increased the amount of time it took to reach this plateau. In the case of MOGA-A, the algorithm was able to identify the minimum possible memory requirement for the AVA benchmark – an optimum value of 90,112 bytes. Eventually, the population in the MOGA configurations consisted almost exclusively of solutions with the same memory requirement. The minimum memory requirement for the SOGA configurations fluctuated during evolution, not converging to a minimum.

We had expected to see that, as the evolution progressed, the SOGA configurations would not experience any significant reduction in memory. Surprisingly, both the average and minimum memory requirements experienced noticeable reductions, albeit not to the same extent as their MOGA counterparts. The

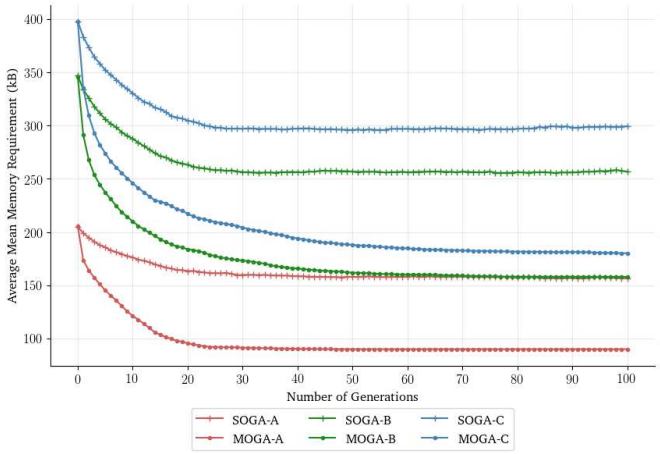

Fig. 3. Average mean memory requirement of all individuals in the population, for each of the six algorithm configurations.

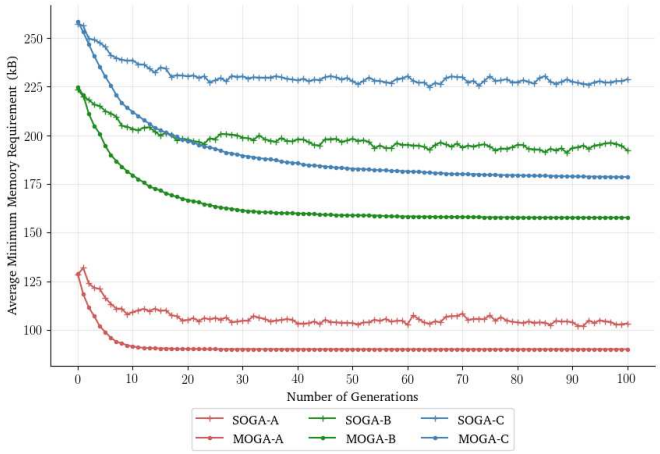

Fig. 5. Average minimum memory requirement of all individuals in the population, for each of the six algorithm configurations.

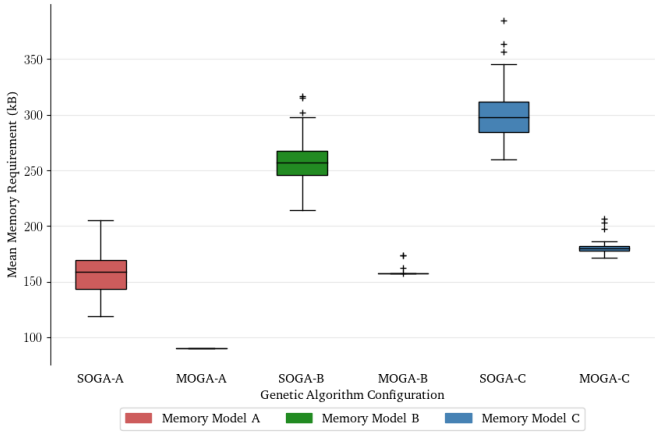

Fig. 4. Distribution of mean memory requirements of all individuals in the final population, for each of the six algorithm configurations.

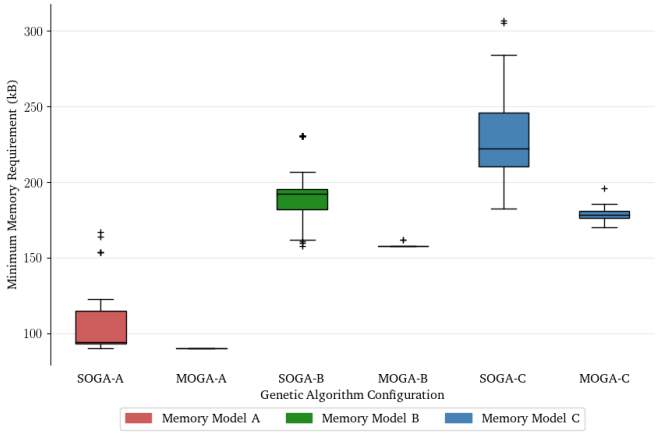

Fig. 6. Distribution of minimum memory requirements of all individuals in the final population, for each of the six algorithm configurations.

AVA benchmark is computationally intensive, and therefore to maintain schedulability, tasks are forced to spread across cores. The memory requirement was reduced as an indirect consequence of this more even distribution. However, recall that we still take into account memory for the communications even if the tasks are mapped to the same core – because communications are copied from one memory space to another. If we elected not to include this (instead using shared memory), then we anticipate that our expectations would have been realised. While it was disappointing to see that the more sophisticated memory models took longer to plateau, this was to be expected. Intuitively, this is because more memory factors are taken into consideration by the fitness function, making it more challenging to find optimal solutions.

The standard deviation of MOGA configurations was unexpectedly low. Since the selection operators for the MOGAs relied on the notion of Pareto dominance, it quickly becomes difficult for weaker individuals to remain in the population, and they are weeded out at an earlier stage. We might attribute the small standard deviation to the fact that increased pressure means that diversity is lost, and the population stagnates. This is not a significant issue, because we still arrive at a satisfactory task mapping and reduced memory requirements.

2) System Schedulability: We next explored system schedulability: the number of unschedulable tasks and flows of all individuals in the population. Once again, the values we report were obtained by averaging over 100 repetitions of the experiment. The plots in Figures 7 and 8 reveal how the mean and minimum number of unschedulable tasks and flows decreased as evolution progressed, respectively.

The results reveal that all six configurations found schedulable solutions early in the evolution process. In other words, the minimum number of unschedulable tasks and flows reached zero quickly. For all memory models, the MOGA configurations appeared initially to converge more quickly to a schedulable solution compared to their SOGA counterparts. However, both SOGAs and MOGAs discovered fully-schedulable solutions by approximately epoch 30. On average, the simpler memory models took less time to schedule in the MOGA configurations. Differences between the models were less apparent when considering the minimum number of unschedulable tasks and flows. We had expected to observe a difference in the amount of time it took the algorithm to find schedulable mappings. Specifically, we anticipated that since the MOGAs

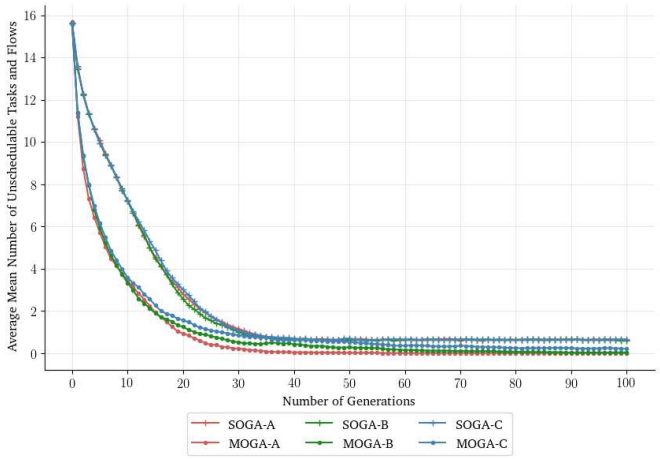

Fig. 7. Average mean number of unschedulable tasks and flows of all individuals in the population, for all of the six algorithm configurations.

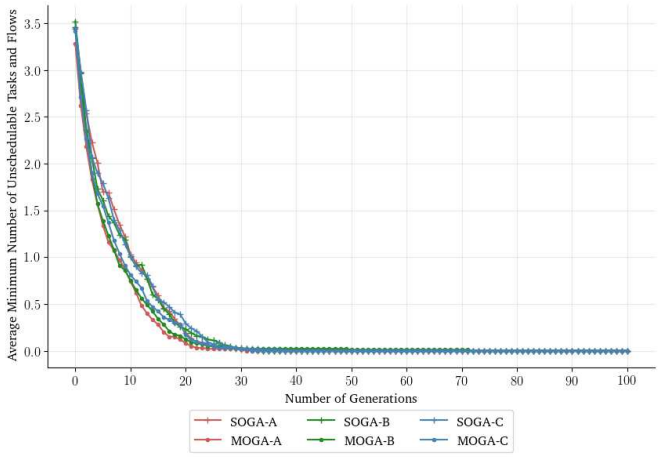

Fig. 8. Average minimum number of unschedulable tasks and flows of all individuals in the population, for all of the six algorithm configurations.

have more factors to consider, these configurations would take extra time. The results we obtained were unexpected. Not only did both the SOGAs and MOGAs find schedulable solutions after approximately the same number of epochs, but taking memory into consideration actually appeared initially to accelerate the rate of convergence. Intuitively, by forcing the tasks to conform to a reduced memory requirement, they are forced to spread out across the NoC earlier on in the evolutionary process. In turn, the schedulability of the system actually improved, likely owing to the fact that the AVA benchmark is computation-bound, rather than communication-bound. Earlier we found that even when the fitness function only considered schedulability, the evolutionary process was able to compress the amount of required memory somewhat. It transpires that the inverse is also true: considering memory as part of the fitness function also has a positive effect on schedulability.

3) Memory Utilisation: We next measured how memory utilisation changed during evolution. Figure 9 shows how the mean memory utilisation of the population varied as evolution progressed, when averaged over 100 independent repetitions.

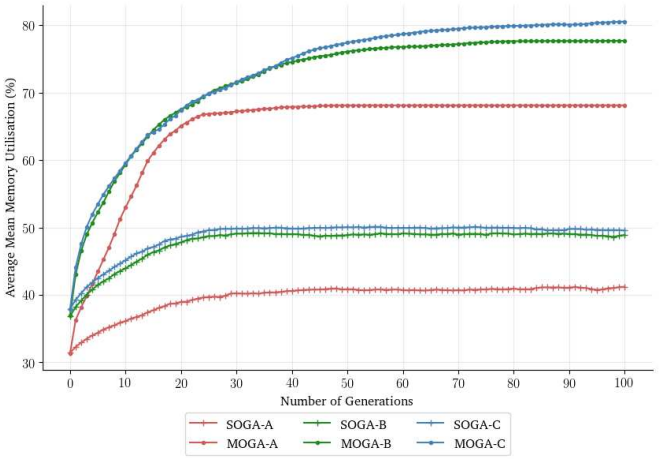

Fig. 9. Average mean memory utilisation of all individuals in the population, for each of the six algorithm configurations.

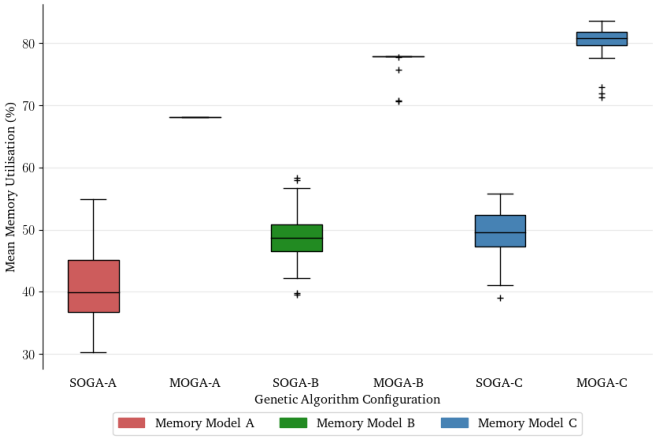

Fig. 10. Distribution of mean memory utilisation of all individuals in the final population, for each of the six algorithm configurations.

Figure 10 shows the distribution of final-generation mean memory utilisations over these repetitions.

Figures 11 and 12 show similar plots to those above. This time, however, we considered the maximum memory utilisation of individuals in the population, rather than taking the average of the whole population.

We found that the mean and maximum memory utilisations followed similar patterns. In the case of the SOGA configurations, the maximum utilisations did not converge to a stable solution and fluctuated in a small range – a behaviour consistent with that seen in other performance metrics. However, in the case of average memory utilisation, we observed that all three of the SOGA configurations followed an identical pattern, with an offset applied to each. It appeared that higher mean and maximum memory utilisations were achieved by memory models B and C. Specifically, employing Model C allowed the algorithm to devise a mapping whose memory utilisation exceeded 84%. The box plot reveals that the MOGAs produced very consistent results in the final round of each repetition, while the SOGAs showed much larger standard deviation.

The results that we obtained were as we expected. Namely, the memory utilisations of mappings produced by the MOGAs

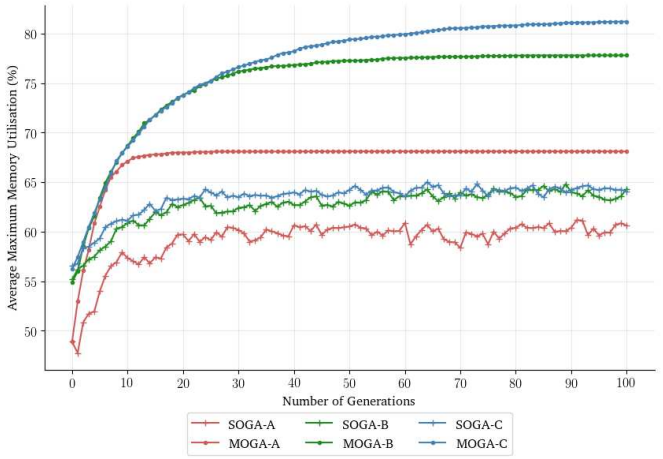

Fig. 11. Average maximum memory utilisation of all individuals in the population, for each of the six algorithm configurations.

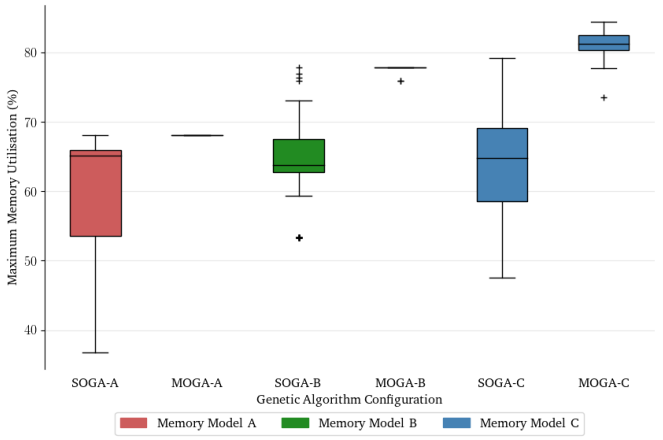

Fig. 12. Distribution of maximum memory utilisation of all individuals in the final population, for each of the six algorithm configurations.

exhibited significantly more efficient memory utilisation, thus wasting less of the available memory. Intuitively, taking memory into account forces tasks with large memory requirements to be distributed over different cores, spreading the load across the system and achieving a more even distribution of memory.

Memory models A and B appear to converge steadfastly to a consistent solution after 100 generations, suggesting that they almost always found the optimal mapping in the case of the AVA benchmark. Memory Model C, however, does not appear to plateau, and therefore the results after 100 generations are slightly more widely distributed. We suspect this is because Memory Model C takes into account code sizes, which are randomly assigned. Stochasticity in the system is likely to manifest as variation in our final results. Running the evolution for more generations would likely result in an eventual plateau.

VI. Conclusions

This paper has provided a solution to the problem of mapping hard real-time tasks onto NoCs under memory constraints. It has proposed three memory models of increasing complexity and realism to better show the impact of such constraints. Experimental work has shown the difficulties imposed by the constraints to a genetic search algorithm increase as the memory models are made more realistic. Nonetheless, the proposed genetic algorithm is capable of finding fully-schedulable mappings that respect these memory constraints. The experiments also uncovered a potential correlation between schedulability and memory optimisation, as the baseline genetic pipeline was able to reduce memory requirements while optimising system schedulability, despite being unaware of that metric. These improvements, however, were of a less significant scale, when compared to the proposed approach.

Further work will address the same metrics over a larger number of benchmarks, and may address more complex memory architectures, such as paged memory models.

References

[1] A. Agarwal. The Tile Processor: A 64-Core Multicore for Embedded Processing. In *11th Annual Workshop on High Performance Embedded Computing (HPEC)*, 2007.

[2] G. Ascia, V. Catania, and M. Palesi. Multi-objective Mapping for Mesh-based NoC Architectures. In *Proc 2nd Int Conf on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 182–187, 2004.

[3] R. Ayari, I. Hafnaoui, G. Beltrame, and G. Nicolescu. Schedulability-guided Exploration of Multi-core Systems. In *Int Symposium on Rapid System Prototyping (RSP)*, 2016.

[4] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti. Guaranteed Services of the NoC of a Manycore Processor. In *Proc Int Workshop on Network on Chip Architectures (NoCArc)*, pages 11–16. ACM, 2014.

[5] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[6] L. S. Indrusiak. End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. *Journal of Systems Architecture*, 60(7):553–561, 2014.

[7] L. S. Indrusiak, A. Burns, and B. Nikolic. Analysis of buffering effects on hard real-time priority-preemptive wormhole networks. *arXiv:1606.02942*, 2016.

[8] H. Kashif, S. Gholamian, and H. Patel. SLA: A Stage-Level Latency Analysis for Real-Time Communication in a Pipelined Resource Model. *IEEE Trans Comput*, 64(4):1177–1190, 2015.

[9] P. Mesidis and L. S. Indrusiak. Genetic mapping of hard real-time applications onto NoC-based MPSoCsa first approach. In *6th Int Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2011.

[10] B. Nikolic, H. I. Ali, S. M. Petters, and L. M. Pinho. Are Virtual Channels the Bottleneck of Priority-aware Wormhole-switched NoC-based Many-cores? In *Proc 21st Int Conf on Real-Time Networks and Systems (RTNS)*, pages 13–22, 2013.

[11] M. Palesi, G. Ascia, F. Fazzino, and V. Catania. Data encoding schemes in networks on chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(5):774–786, 2011.

[12] M Norazizi Sham Mohd Sayuti and Leandro Soares Indrusiak. Real-Time Low-Power Task Mapping in Networks-on-Chip. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 14–19, 2013.

[13] Z. Shi and A. Burns. Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching. In *IEEE/ACM NOCS Symposium*, pages 161–170, 2008.

[14] Z. Shi, A. Burns, and L. S. Indrusiak. Schedulability Analysis for Real Time On-Chip Communication with Wormhole Switching. *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 1(2), 2010.

[15] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proc 50th IEEE/ACM Design Automation Conference (DAC)*, 2013.

[16] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: an NP-Hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[17] Q. Xiong, F. Wu, Z. Lu, and C. Xie. Extending Real-Time Analysis for Wormhole NoCs. *IEEE Trans Comput*, 66(9):1532–1546, 2017.